

JPEG Encoder

Comprehensive Implementation Guide

From Pixels to Compressed Bytes

A Deep Dive into DCT, Quantization, and Huffman Coding

Python Implementation with Mathematical Foundation

December 25, 2025

Sami Hindi

Contents

1	Introduction	2
1.1	Encoding Pipeline Overview	2
2	Color Space Conversion	3
2.1	Mathematical Formulation	3
2.2	Implementation	3
3	Chroma Subsampling	4
3.1	Subsampling Modes	4
4	Discrete Cosine Transform (DCT)	5
4.1	2D-DCT Formula	5
4.2	DCT Basis Functions	5
4.3	Matrix Implementation	6
4.4	Implementation	6
5	Quantization	7
5.1	Quantization Formula	7
5.2	Standard Quantization Tables	7
5.3	Quality Scaling	7
6	Zigzag Scanning & Run-Length Encoding	8
6.1	Zigzag Pattern	8
6.2	Run-Length Encoding	8
7	Huffman Encoding	9
7.1	DC Coefficient Encoding	9
7.2	Standard DC Luminance Huffman Table	9
7.3	Bit Writing with Byte Stuffing	9
8	JPEG File Structure	11
8.1	Marker Overview	11
8.2	SOF0 Component Specification	11
9	Implementation Results	12
9.1	Compression Performance	12
9.2	Quality Metrics	12
10	Usage Guide	13
10.1	Command Line	13
10.2	Python API	13
11	References	13

1 Introduction

What is JPEG?

JPEG (Joint Photographic Experts Group) is a widely-used lossy compression standard for digital images. It achieves significant file size reduction by exploiting characteristics of human vision—we are more sensitive to brightness changes than color changes, and less sensitive to high-frequency details.

The JPEG algorithm consists of several key stages:

1. **Color Space Conversion** — RGB to YCbCr
2. **Chroma Subsampling** — Reduce color resolution
3. **Block Splitting** — Divide into 8×8 blocks
4. **DCT Transform** — Convert to frequency domain
5. **Quantization** — Reduce precision (lossy step)
6. **Entropy Coding** — Huffman compression

1.1 Encoding Pipeline Overview

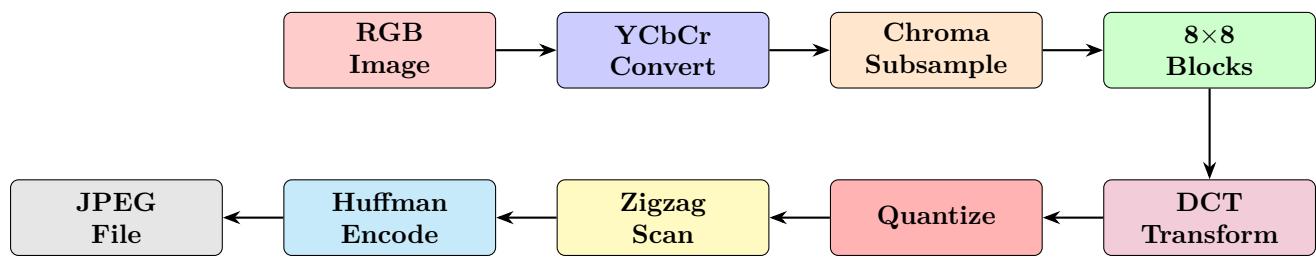


Figure 1: JPEG encoding pipeline showing the transformation from RGB input to compressed output.

2 Color Space Conversion

RGB to YCbCr Transformation

Human vision is more sensitive to luminance (brightness) than chrominance (color). JPEG exploits this by separating the image into:

- **Y** — Luminance (brightness)
- **C_b** — Blue-difference chroma component
- **C_r** — Red-difference chroma component

2.1 Mathematical Formulation

The ITU-R BT.601 standard defines the conversion:

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.5 \\ 0.5 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \quad (1)$$

Definition

The coefficients in the Y row (0.299, 0.587, 0.114) reflect human perception—we are most sensitive to green, then red, then blue.

2.2 Implementation

```

1 def rgb_to_ycbcr(rgb: np.ndarray) -> np.ndarray:
2     """Convert RGB to YCbCr color space."""
3     transform = np.array([
4         [0.299, 0.587, 0.114],
5         [-0.169, -0.331, 0.5],
6         [0.5, -0.419, -0.081]
7     ])
8     ycbcr = np.zeros_like(rgb, dtype=np.float64)
9     # Apply matrix transformation
10    for i in range(3):
11        ycbcr[:, :, i] = sum(transform[i, j] * rgb[:, :, j]
12                                for j in range(3))
13    ycbcr[:, :, 1:] += 128 # Offset chroma
14    return np.clip(ycbcr, 0, 255)

```

3 Chroma Subsampling

Reducing Color Resolution

Since human vision has lower spatial resolution for color than brightness, we can reduce the resolution of the Cb and Cr channels with minimal perceptual impact.

3.1 Subsampling Modes

4:4:4 (No subsampling)

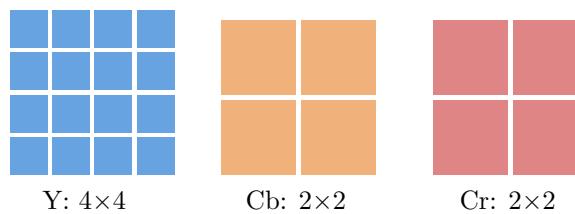
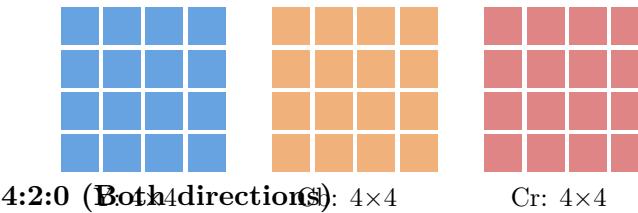


Figure 2: Comparison of chroma subsampling modes. 4:2:0 reduces chroma data to 25% of original.

Table 1: Subsampling Mode Comparison

Mode	H Ratio	V Ratio	Data Reduction
4:4:4	1:1	1:1	None
4:2:2	2:1	1:1	33%
4:2:0	2:1	2:1	50%

4 Discrete Cosine Transform (DCT)

Frequency Domain Transformation

The DCT converts spatial pixel values into frequency coefficients. Low frequencies (upper-left of the block) represent smooth gradients, while high frequencies (lower-right) represent sharp edges and details.

4.1 2D-DCT Formula

The two-dimensional DCT-II for an $N \times N$ block is defined as:

$$F(u, v) = \frac{1}{4} C(u) C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) \cos \left[\frac{(2x+1)u\pi}{2N} \right] \cos \left[\frac{(2y+1)v\pi}{2N} \right] \quad (2)$$

where:

$$C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } k = 0 \\ 1 & \text{otherwise} \end{cases} \quad (3)$$

Definition

The coefficient $F(0, 0)$ is called the **DC coefficient** and represents the average value of the block. All other coefficients are **AC coefficients**.

4.2 DCT Basis Functions

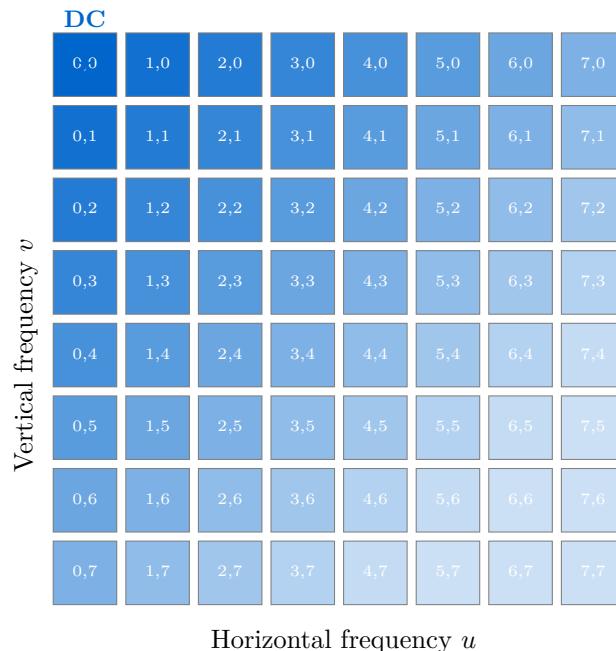


Figure 3: The 64 DCT basis function positions. (u, v) indicates frequency indices. Upper-left $(0,0)$ is DC; frequency increases toward lower-right.

4.3 Matrix Implementation

For efficiency, the 2D-DCT is computed using matrix multiplication:

$$\mathbf{F} = \mathbf{D} \cdot \mathbf{f} \cdot \mathbf{D}^T \quad (4)$$

where \mathbf{D} is the DCT transformation matrix:

$$D_{ij} = \begin{cases} \frac{1}{\sqrt{N}} & \text{if } i = 0 \\ \sqrt{\frac{2}{N}} \cos \left[\frac{(2j+1)i\pi}{2N} \right] & \text{otherwise} \end{cases} \quad (5)$$

4.4 Implementation

```

1  def create_dct_matrix(n: int = 8) -> np.ndarray:
2      """Create the DCT-II transformation matrix."""
3      dct_matrix = np.zeros((n, n), dtype=np.float64)
4      for i in range(n):
5          for j in range(n):
6              if i == 0:
7                  dct_matrix[i, j] = 1 / np.sqrt(n)
8              else:
9                  dct_matrix[i, j] = np.sqrt(2/n) * \
10                     np.cos((2*j + 1) * i * np.pi / (2*n))
11      return dct_matrix
12
13 DCT_MATRIX = create_dct_matrix(8)
14
15 def dct_2d(block: np.ndarray) -> np.ndarray:
16     """Apply 2D DCT: F = D * f * D^T"""
17     return DCT_MATRIX @ block @ DCT_MATRIX.T

```

5 Quantization

The Lossy Step

Quantization is where JPEG achieves most of its compression—and where information is permanently lost. Each DCT coefficient is divided by a quantization value and rounded to an integer.

5.1 Quantization Formula

$$F_Q(u, v) = \text{round} \left(\frac{F(u, v)}{Q(u, v)} \right) \quad (6)$$

5.2 Standard Quantization Tables

Luminance (Y)

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
112	295	981	1121	1001	1039	999	99pt

Chrominance (Cb, Cr)

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Figure 4: Standard JPEG quantization tables (ITU-T T.81). Higher values = more compression.

5.3 Quality Scaling

The quality parameter (1–100) scales the quantization table:

$$Q_{\text{scaled}}(u, v) = \text{clip} \left(\left\lfloor \frac{Q(u, v) \cdot S + 50}{100} \right\rfloor, 1, 255 \right) \quad (7)$$

$$\text{where } S = \begin{cases} \frac{5000}{q} & \text{if } q < 50 \\ 200 - 2q & \text{if } q \geq 50 \end{cases}$$

6 Zigzag Scanning & Run-Length Encoding

Preparing for Entropy Coding

After quantization, many high-frequency coefficients become zero. The zigzag scan orders coefficients from low to high frequency, grouping zeros together for efficient run-length encoding.

6.1 Zigzag Pattern

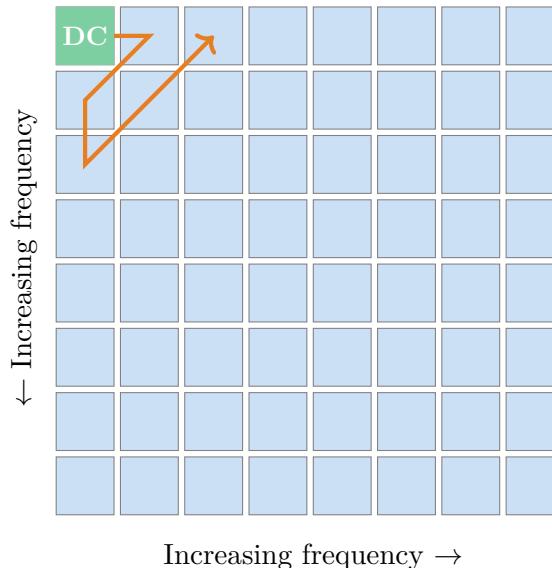


Figure 5: Zigzag scan begins at DC (green) and proceeds toward high frequencies.

6.2 Run-Length Encoding

AC coefficients are encoded as (run, value) pairs:

Algorithm: RLE for AC Coefficients

1. Start after DC coefficient (position 1)
2. Count consecutive zeros (run length)
3. Encode: (run_length, size, value)
4. If run > 15: emit ZRL symbol (15, 0) and continue
5. End with EOB symbol (0, 0) if trailing zeros

Definition

Example: Coefficients [DC, 5, 0, 0, -3, 0, 0, 0, 0, 1, 0...0]
Encoded as: (0,3,5), (2,2,-3), (4,1,1), EOB

7 Huffman Encoding

Entropy Coding

Huffman coding assigns shorter bit sequences to more frequent symbols. JPEG uses separate tables for DC and AC coefficients, and for luminance and chrominance.

7.1 DC Coefficient Encoding

DC coefficients are encoded **differentially**—each DC value is the difference from the previous block’s DC:

$$\text{DIFF} = \text{DC}_n - \text{DC}_{n-1} \quad (8)$$

The difference is encoded as:

1. Huffman code for the **size** (number of bits needed)
2. The actual **value** in that many bits

7.2 Standard DC Luminance Huffman Table

Table 2: DC Luminance Huffman Codes (Partial)

Size	Code Length	Code (binary)	Value Range
0	2	00	0
1	3	010	-1, 1
2	3	011	-3.. - 2, 2..3
3	3	100	-7.. - 4, 4..7
4	3	101	-15.. - 8, 8..15
5	3	110	-31.. - 16, 16..31
6	4	1110	-63.. - 32, 32..63

7.3 Bit Writing with Byte Stuffing

Warning

JPEG requires **byte stuffing**: whenever `0xFF` appears in the entropy-coded data, it must be followed by `0x00` to distinguish it from markers.

```
1 class BitWriter:
2     def write_bits(self, value: int, num_bits: int):
3         self.bit_buffer = (self.bit_buffer << num_bits) | value
4         self.bit_count += num_bits
5
6         while self.bit_count >= 8:
7             self.bit_count -= 8
8             byte = (self.bit_buffer >> self.bit_count) & 0xFF
9             self.buffer.append(byte)
10            # Byte stuffing: 0xFF -> 0xFF 0x00
11            if byte == 0xFF:
12                self.buffer.append(0x00)
```

8 JPEG File Structure

JFIF Format

A JPEG file consists of segments, each beginning with a two-byte marker. The JFIF (JPEG File Interchange Format) is the most common container format.

8.1 Marker Overview

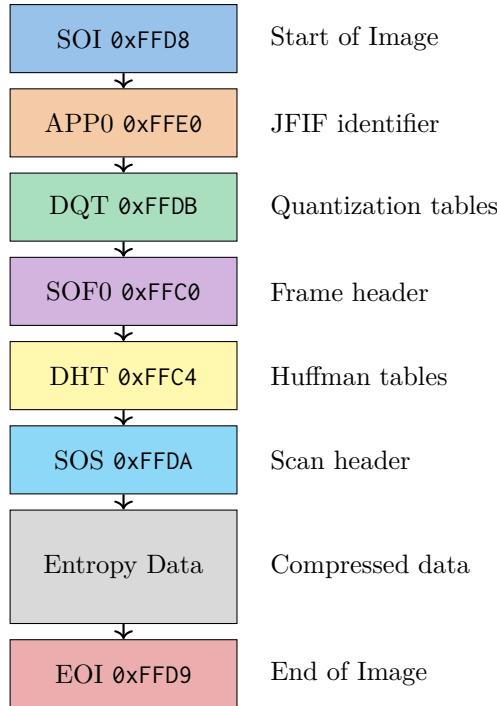


Figure 6: JPEG file structure showing marker sequence.

8.2 SOF0 Component Specification

Table 3: SOF0 Component Parameters for Different Subsampling Modes

Component	ID	4:4:4	4:2:2	4:2:0
Y (Luminance)	1	0x11	0x21	0x22
Cb (Chroma)	2	0x11	0x11	0x11
Cr (Chroma)	3	0x11	0x11	0x11

9 Implementation Results

9.1 Compression Performance

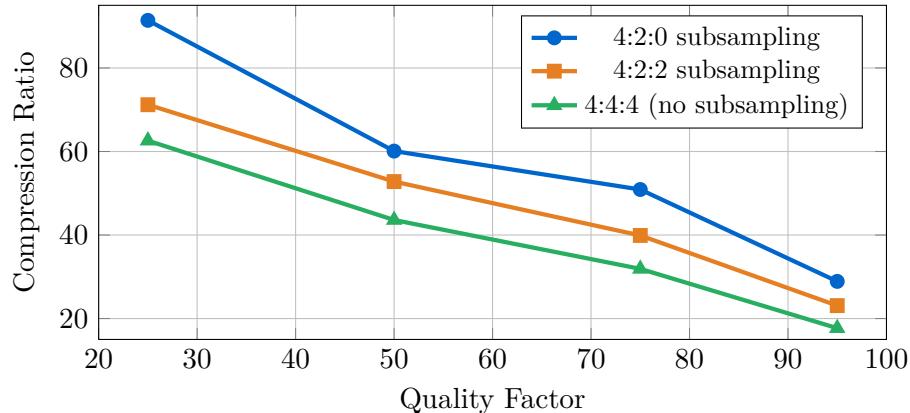


Figure 7: Compression ratio vs. quality factor for different subsampling modes.

9.2 Quality Metrics

Table 4: PSNR Results for 256×256 Test Image

Quality	4:2:0 PSNR	4:2:2 PSNR	4:4:4 PSNR
25	30.9 dB	33.4 dB	39.1 dB
50	31.9 dB	34.4 dB	41.9 dB
75	32.1 dB	35.1 dB	48.4 dB
95	32.1 dB	35.2 dB	52.0 dB

Definition

PSNR (Peak Signal-to-Noise Ratio) is calculated as:

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{255^2}{\text{MSE}} \right) \text{ dB} \quad (9)$$

Higher values indicate better quality. Values > 40 dB are considered excellent.

10 Usage Guide

10.1 Command Line

```
# Basic encoding
python jpeg.py input.png output.jpg

# High quality, no subsampling
python jpeg.py input.png output.jpg -q 95 -s 444

# Generate test image
python jpeg.py -test -v
```

10.2 Python API

```
1 from jpeg import encode_image, save_jpeg, SubsamplingMode
2 import numpy as np
3
4 # Load or create image
5 image = np.random.randint(0, 256, (512, 512, 3), dtype=np.uint8)
6
7 # Encode to bytes
8 jpeg_bytes = encode_image(image, quality=85)
9
10 # Save to file
11 save_jpeg(image, "output.jpg",
12             quality=90,
13             subsampling=SubsamplingMode.MODE_420)
```

11 References

1. **ITU-T T.81** — Digital compression and coding of continuous-tone still images: Requirements and guidelines (JPEG standard)
2. **ITU-R BT.601** — Studio encoding parameters of digital television
3. Wallace, G.K. (1991). *The JPEG Still Picture Compression Standard*. Communications of the ACM.

Implementation Complete

Full source code available in `jpeg.py`